Midterm II Exam

15-122 Principles of Imperative Computation
Frank Pfenning

March 31, 2011

Name: **Sample Solution** Andrew ID: **fp** Section:

Instructions

- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 4 problems.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.
- Consider writing out programs or diagrams on scratch paper first.

	Unbounded Arrays	Heaps	BSTs	Rotations	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score	**30**	**25**	**25**	**20**	**100**
Max	30	25	25	20	100
Grader	??	??	??	??	

1. Unbounded Arrays (30 pts)

In this problem we reconsider unbounded arrays, as introduced in lecture. Recall the basic idea: we begin by allocating a fixed size array, adding new elements at the end. When the array is full, we create a new array of twice size of the old array, copy the existing elements over, and continue to work with the new array. In our analysis we count the number of write operations to arrays.

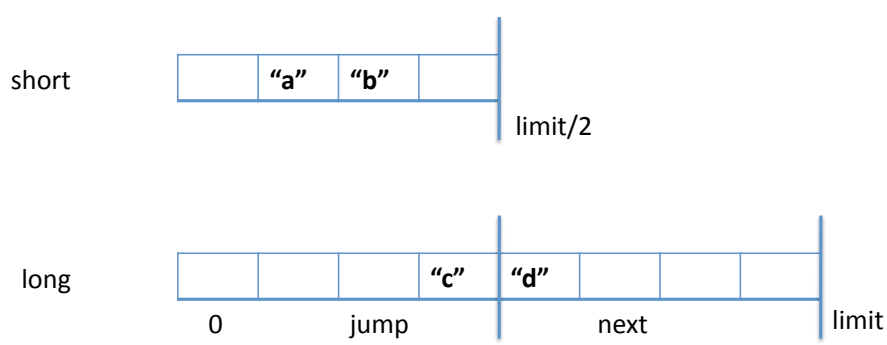
Task 1 (2 pts). What is a worst-case bound on the number of array writes that have to be performed during a *single* insert operation, assuming there are already n elements in the unbounded array? Express your answer precisely in terms of n and not in big-O notation. You do not need to justify your answer.

n+1

Task 2 (3 pts). What is a worst-case bound on the total number of array writes that have to be performed during a sequence of n insert operations, starting from an empty array. Express your answer precisely in terms of n and not in big-O notation. You do not need to justify your answer; we obtained it in lecture by amortized analysis.

3*n

Next we discuss a technique to eliminate the potentially long pauses when we have to double the size of the array. Instead of copying the whole array at once, we copy it element by element. For this to work we need two arrays: a short one and a long one. The elements of the unbounded array are distributed over these two arrays. We only ever add elements to the long array. Whenever we do so, we also copy one element from the short array to the long array. Besides the usual index *next* that indicates the next insertion point, we also maintain an index *jump* where the lookup function "jumps" from the short to the long array. For example, after adding strings `"a"` through `"d"` to the empty array, we are in the following situation, where blank array elements are irrelevant.



If we now add `"e"` we also copy `"b"` from the short to the long array, adjusting indices as appropriate. In order to simplify boundary conditions, we do not use index 0 of the underlying arrays, but start at index 1.

Now to the implementation. All the crucial invariants are expressed in the comments below. You may assume that they are checked in the `is_uba` function. **Read these invariants carefully.**

```
struct uba {
  int limit;                    /* limit > 0 */
  int next;                     /* 1 <= next && next <= limit */
  int jump;                     /* jump + next == limit-1 */
  elem[] short;                 /* \length(short) = limit/2 */
  elem[] long;                  /* \length(long) = limit */
};
typedef struct uba* uba;
bool is_uba(uba L);
```

Task 3 (5 pts). Complete the following function that returns the element at index i.

```
elem uba_get(uba L, int i)
//@requires is_uba(L);
//@requires 1 <= i && i < L->next;
{
  if (i > L->jump)
    return L->long[i];
  else
    return L->short[i];
}
```

Task 4 (3 pts). Complete the following function to double the size of the unbounded array. [**Hint:** let yourself be guided by the data structure invariants.]

```
void uba_double(uba L)
//@requires is_uba(L);
//@ensures is_uba(L);
{
  L->limit = 2*L->limit;
  L->short = L->long;
  L->long = alloc_array(elem, L->limit);

  L->jump = L->next - 1 ;
  return;
}
```

Task 5 (12 pts). Complete the following function to add a new element to the end of the array. [**Hint:** let yourself be guided by the data structure invariants.]

```
void addend(uba L, elem e)
//@requires is_uba(L);
//@ensures is_uba(L);
{
  if (L->next == L->limit)
    uba_double(L);
  L->long[L->next] = e;
  L->next++;
  L->long[L->jump] = L->short[L->jump];
  L->jump--;
}
```

Task 6 (2 pts). In the worst case, how many array write operations do we have to perform for a single insertion (function `addend`)? Again, give your answer directly, not in big-O notation.

2 (`uba_double` does not write to any array)

Task 7 (3 pts). Would this incremental copying technique work for hash tables that dynamically adjust their size? Justify your answer in one or two sentences.

No, at least not in a straightforward way. That's because we have to rehash all elements before inserting them into the larger table, so we wouldn't reliably find the elements we haven't moved yet.

2. Heaps (25 pts)

In this problem we explore the representation of unbounded priority queues as pointer-based trees, instead using arrays. We use the following type:

```
struct tree {
  int priority;
  int size;
  struct tree* left;
  struct tree* right;
};
typedef struct tree* tree;
```

See Task 3 for an explanation of the `size` field.

Task 1 (2 pts). State the ordering invariant for min-heaps, as discussed in lecture.

For every node in the tree, its priority is less or equal to the priority of all of its children.

Task 2 (8 pts). Write a function `heap_ord` to check the ordering invariant for heaps, represented as trees. [**Hint:** Think recursively.]

```
bool heap_ord(tree T) {
  if (T == NULL) return true;
  if (T->left != NULL && T->left->priority < T->priority)
    return false;
  if (T->right != NULL && T->right->priority < T->priority)
    return false;
  return heap_ord(T->left) && heap_ord(T->right);
}
```

In order to keep the heap as balanced as possible, we maintain a size field in each node. It represents the total number of elements in the tree with that node as its root. We have the following balance invariant that *replaces* the old shape invariant:

> **Heap Balance Invariant:** The number of elements in the left and right subtrees at every node differ by at most 1.

Assume the following functions:

```
bool heap_bal(tree T);  /* check if tree is heap-balanced, O(n) */
tree leaf(int x);       /* construct a leaf node with priority x, O(1) */
int heap_size(tree T);  /* return size of tree, O(1) */
```

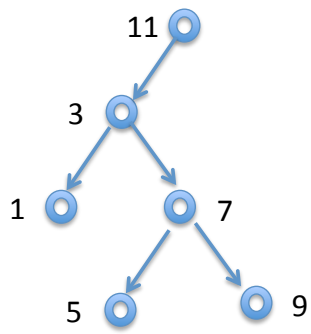
Task 3 (15 pts). Write a recursive function `heap_insert` that inserts a new element into the heap, *maintaining both order and balance invariants*. It should have asymptotic complexity $O(\log(n))$ where $n = heap_size(T)$. [**Hint:** Keep in mind that heaps are very different from binary search trees.]

```
tree heap_insert(tree T, int x)
//@requires heap_ord(T) && heap_bal(T);
//@ensures heap_ord(\result) && heap_bal(\result);
{
  if (T == NULL) return leaf(x);
  if (T->priority > x) {
    /* swap x with priority at T */
    int tmp = T->priority;
    T->priority = x;
    x = tmp;
  }
  //@assert x >= T->priority;
  if (heap_size(T->left) <= heap_size(T->right))
    T->left = heap_insert(T->left, x);
  else
    T->right = heap_insert(T->right, x);
  T->size++;
  return T;
}
```

3. Binary Search Trees (25 pts)

In this problem we consider plain binary search trees, without rebalancing.

Task 1 (3 pts). Consider the tree below. Give a sequence of numbers that, when inserted in the given order into an empty tree, would construct exactly this tree.



11, 3, 1, 7, 5, 9

Task 2 (5 pts). Assume that we have a sorted array from which we want to construct a binary search tree. Which order of insertion should we follow to obtain a tree that is as balanced as possible?

To insert a range $A[lower..upper)$ of an array, start with the middle element $A[mid]$ for $mid = lower + (upper - lower)/2$ and then recursively insert the ranges $A[lower..mid)$ and $A[mid + 1..upper)$, in either order.

We are now working with the following types.

```
struct tree {
  elem data;
  struct tree* left;
  struct tree* right;
};
typedef struct tree* tree;
struct bst {
  tree root;
  int size;        /* size == tree_size(root), see Task 3 */
};
typedef struct bst* bst;
```

Task 3 (5 pts). Write a function `tree_size` to compute the total number of elements in a tree.

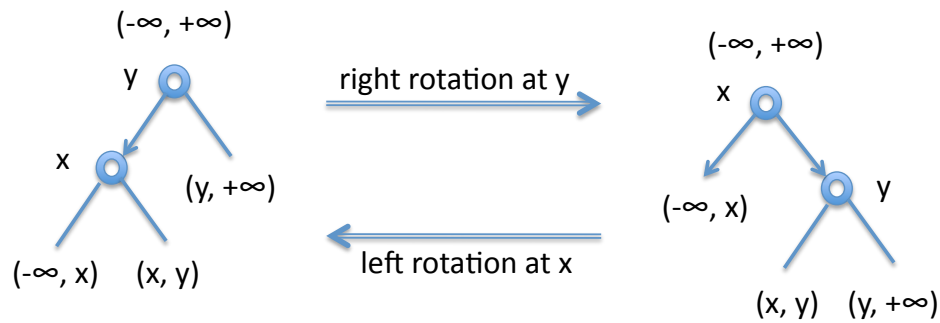
```
int tree_size(tree T) {
  if (T == NULL)
    return 0;
  else
    return tree_size(T->left)+tree_size(T->right)+1;
}
```

Task 4 (12 pts). Next you are given some code to transform a binary search tree to a sorted array of elements. It is your job to deduce how it works. Fill in pre- and post-conditions in the `tree2array` function below. Your annotations should at least be strong enough to verify that no null pointer dereference or out-of-bounds array access is possible. In addition to `tree_size` you may also use `is_sorted` and `is_ordtree` declared below.

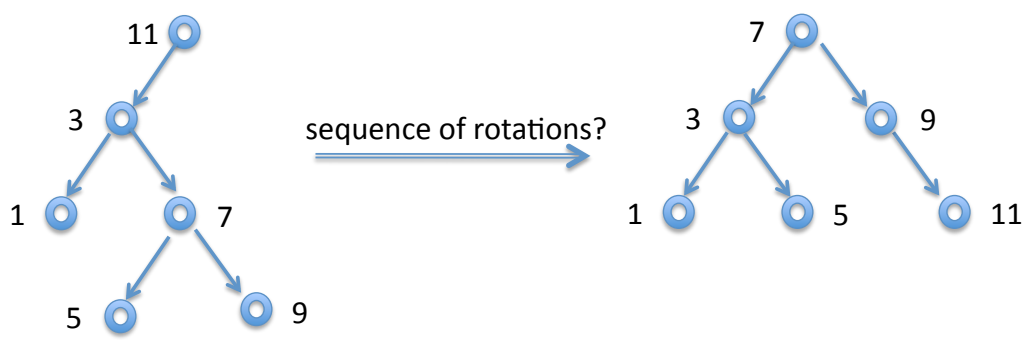
```
bool is_sorted(elem[] A, int lower, int upper) /* A[lower..upper) is sorted */
//@requires 0 <= lower && lower <= upper && upper <= \length(A);
  ;
bool is_ordtree(tree T);                    /* binary search tree ordering */

int tree2array(tree T, elem[] A, int i)
//@requires is_ordtree(T);
//@requires 0 <= i && i <= \length(A) - tree_size(T);
//@ensures \result == i + tree_size(T);
//@ensures is_sorted(A, i, \result);
{
 if (T == NULL) return i;
  int j = tree2array(T->left, A, i);
  A[j] = T->data;
  return tree2array(T->right, A, j+1);
}
```

4. Rotations (20 pts)

Task 1 (5 pts). Complete the picture below with the missing tree on the right.



Task 2 (10 pts). Give a sequence of single left or right rotations, and where they are performed, to transform the tree on the left to the tree on the right. For full credit, find the shortest sequence (which requires less than 5 steps). Indicate in the rightmost column if the tree *after* the rotation satisfies the balance invariant of the AVL trees.



Step	Left or Right?	at	AVL?
1	left	3	no
2	right	11	yes
3	right	11	yes
4			
5			

Task 3 (5 pts). Is it possible to transform any binary search tree to any other binary search tree that has exactly the same elements? If not, show a counterexample. If yes, briefly explain how.

Yes. First, you identify the root of the target tree in the source and move it to the top with a sequence of left and right rotations. Now the left and right subtrees must have exactly the same elements as the left and right subtrees of the target, so we can transform them recursively into the desired shape. The empty tree is transformed to the empty tree by zero rotations.